The Stanford `libcs106` library, Academic Year 2024-25

`#include "map.h"`

`class Map<KeyType, ValueType>`

This class maintains an association between ***keys*** and ***values***. The types used for keys and values are specified using templates, which makes it possible to use this structure with any data type.

The map uses a binary search tree (BST) structure internally. Because of this choice of internal representation, the **`KeyType`** for the keys stored in a **`Map`** must define a natural ordering through a **`less`** function and/or **`<`** operator so that the keys can be compared and ordered. The **`ValueType`** does not need to provide any such natural ordering. The range-based for loop will iterate over the map keys in sorted order. The Map operations to add/access/remove an entry run in O(logN) time. .

Constructor

`Map()`	O(1)	Initializes a new empty map that associates keys and values of the specified types.

Methods

`clear()`	O(N)	Removes all entries from this map.
`containsKey(`*`key`*`)`	O(log N)	Returns **`true`** if there is an entry for **`key`** in this map.
`equals(`*`map`*`)`	O(N)	Returns **`true`** if the two maps contain the same entries.
`firstKey()`	O(1)	Returns the first key in this map in the order established by a for-each loop.
`get(`*`key`*`)`	O(log N)	Returns the value associated with **`key`** in this map.
`isEmpty()`	O(1)	Returns **`true`** if this map contains no entries.
`keys()`	O(N)	Returns a **`Vector`** copy of all keys in this map.
`lastKey()`	O(log N)	Returns the last value in this map in the order established by a for-each loop.
`mapAll(`*`fn`*`)`	O(N)	Iterates through the map entries and calls **`fn(key, value)`** for each one.

`put(key, value)`	O(log N)	Associates **`key`** with **`value`** in this map.
`remove(key)`	O(log N)	Removes any entry for **`key`** from this map.
`size()`	O(1)	Returns the number of entries in this map.
`toString()`	O(N)	Returns a printable string representation of this map.
`values()`	O(N)	Returns a **`Vector`** copy of all values in this map.

Operators

`for (KeyType key : map)`	O(N)	Iterates through the keys in a map.
`map[key]`	O(log N)	Selects the value associated with **`key`**.
`map1 == map1`	O(N)	Returns **`true`** if **`map1`** and **`map2`** contain the same entries.
`map1 != map2`	O(N)	Returns **`true`** if **`map1`** and **`map2`** are different.
`map1 + map2`	O(NlogN)	Creates a new map which contains all **`map1`** entries added to all **`map2`** entries.
`map1 += map2`	O(NlogN)	Adds all **`map2`** entries to **`map1`**.
`map1 - map2`	O(NlogN)	Creates a new map which contains all **`map1`** entries minus all **`map2`** entries.
`map1 -= map2`	O(NlogN)	Removes all **`map2`** entries from **`map1`**.
`map1 * map2`	O(NlogN)	Creates a new map which contains all entries that appear in both **`map1`** and **`map2`**.
`map1 *= map2`	O(NlogN)	Removes any entries from **`map1`** that are not present in **`map2`**.
`ostream << map`	O(N)	Outputs the contents of the map to the given output stream.
`istream >> map`	O(N log N)	Reads the contents of the given input stream into the map.

Constructor detail

`Map();`

Initializes a new empty map that associates keys and values of the specified types. You may also optionally provide an initializer list of key-value pairs. The newly created map will contain those entries.

Usage:

```
Map<KeyType, ValueType> map;
Map<KeyType, ValueType> map = {{ k1, v1}, { k2, v2 }};
```

Method detail

`void clear();`

Removes all entries from this map.

Usage:

```
map.clear();
```

`bool containsKey(const KeyType& key) const;`

Returns **`true`** if there is an entry for **`key`** in this map.

Usage:

```
if (map.containsKey(key)) ...
```

`bool equals(const Map& map) const;`

Returns **`true`** if the two maps contain exactly the same key/value pairs. Identical in behavior to the **==** operator.

Usage:

```
if (map1.equals(map2)) ...
```

`KeyType firstKey() const;`

Returns the first key in the map in the order established by a for-each loop. If map is empty, **`firstKey`** signals an error.

Usage:

```
KeyType first = map.firstKey();
```

`ValueType get(const KeyType& key) const;`

Returns the value associated with **`key`** in this map. If **`key`** is not found, **`get`** returns the default value for **`ValueType`**.

Usage:

```
ValueType value = map.get(key);
```

`bool isEmpty() const;`

Returns **`true`** if this map contains no entries.

Usage:

```
if (map.isEmpty()) ...
```

`Vector<KeyType> keys() const;`

Returns a **`Vector`** copy of all keys in this map. The keys will appear in the same order that a for-each loop over the map would produce them. Because a map cannot contain duplicate keys, the elements of the vector will be unique.

Usage:

```
Vector<KeyType> keys = map.keys();
```

`KeyType lastKey() const;`

Returns the last key in the map in the order established by a for-each loop. If map is empty, **`lastKey`** signals an error.

Usage:

```
KeyType last = map.lastKey();
```

`void mapAll(std::function<void (const KeyType&, const ValueType&)> fn) const;`

Iterates through the map entries and calls **`fn(key, value)`** for each one. The keys are processed in ascending order, as defined by the comparison function.

Usage:

```
map.mapAll(fn);
```

`void put(const KeyType& key, const ValueType& value);`

Associates **`key`** with **`value`** in this map. Any previous value associated with **`key`** is replaced by the new value.

Usage:

```
map.put(key, value);
```

`void remove(const KeyType& key);`

Removes any entry for **`key`** from this map.

Usage:

```
map.remove(key);
```

```
int size() const;
```

Returns the number of entries in this map.

Usage:

```
int nEntries = map.size();
```

```
string toString() const;
```

Returns a printable string representation of this map. such as `"{k1:v1, k2:v2, k3:v3}"`. The key/value pairs will be listed in ascending order by key.

Usage:

```
string str = map.toString();
```

```
Vector<ValueType> values() const;
```

Returns a `Vector` copy of all values in this map. The values will appear in the same order that a for-each loop over the map would produce them. A map can contain duplicate values, so the elements of the vector are not guaranteed to be unique.

Usage:

```
Vector<ValueType> values = map.values();
```

Operator detail

```
for (KeyType key : map)
```

The range-based for loop can be used to iterate through the elements in a collection. The iteration accesses map keys in ascending order. An error is signaled if you attempt to add/remove elements from a collection while iterating over it.

Usage:

```
for (KeyType key : map) {
        cout << key << " = " << map[key] << endl;
}
```

```
ValueType& operator[](const KeyType& key);
const ValueType& operator[](const KeyType& key) const;
```

Selects the value associated with `key`. This syntax makes it easy to think of a map as an "associative array" indexed by the key type. If `key` is already present in the map, this function returns a reference to its associated value. If key is not present in the map, a new entry is created whose value is set to the default for the value type.

Note: `get` and `operator[]` have a small but significant difference when used to retrieve the value for a key not contained in the map. Both expressions return the default value, but accessing `map[key]` adds this new entry to the map while `map.get(key)` does not.

Usage:

```
map[key]
```

```
Map operator+(const Map& map2) const;
```

Creates a new map which combines the entries from `map1` and `map2`.

Usage:

```
map1 + map2
```

```
Map operator*(const Map& map2) const;
```

Creates a new map which contains those entries that appear in both `map1` and `map2`.

Usage:

```
map1 * map2
```

`Map operator-(const Map& map2) const;`

Creates a new map which is the difference of the entries in `map1` minus those in `map2`.

Usage:

```
map1 - map2
```

`Map& operator+=(const Map& map2);`

Adds all of the entries from `map2` to `map1`.

Usage:

```
map1 += map2;
```

`Map& operator*=(const Map& map2);`

Removes any entries from `map1` that are not present in `map2`.

Usage:

```
map1 *= map2;
```

`Map& operator-=(const Map& map2);`

Removes the entries in `map2` from `map1`.

Usage:

```
map1 -= map2;
```

`ostream& operator<<(const Map& map);`

Outputs the contents of **`map`** to the given output stream. The output is in the form **`{k1:v1, k2:v2, k3:v3}`**. The entries will be listed in ascending order of key.

Usage:

```
cout << map << endl;
```

`istream& operator>>(Map& map);`

Reads the contents of the given input stream into **`map`**. Any previous contents of the map are replaced. The input is expected to be in the form **`{k1:v1, k2:v2, k3:v3}`**. If unable to read a proper map from the stream, the operation results in a stream fail state.

Usage:

```
if (infile >> map) ...
```
